4

Sea la siguiente solución al problema del producto de matrices de nxn con P procesos trabajando en paralelo.

```
process worker[w = 1 to P] {                  # strips en paralelo (p strips de n/P filas) }
        int first = (w-1) * n/P;               # Primera fila del strip
        int last = first + n/P - 1;            # Última fila del strip
        for [i = first to last] {
            for [j = 0 to n-1] {
                c[i,j] = 0.0;
                for [k = 0 to n-1]
                    c[i,j] = c[i,j] + a[i,k]*b[k,j];
            }
        }
}
```

a) **Suponga que n=128 y cada procesador es capaz de ejecutar un proceso. ¿Cuántas asignaciones, sumas y productos se hacen secuencialmente (caso en que P=1)?**

Si el algoritmo se ejecuta secuencialmente se tienen:
Asignaciones: $128^3 + 128^2 = 2097152 + 16384 = 2113536$
Sumas: $128^3 = 2097152$.
Productos: $128^3 = 2097152$.

b) **¿Cuántas se realizan en cada procesador en la solución paralela con P=8?**

Si tenemos 8 procesos cada uno con un strip de 16 (128/8) los cálculos de tiempo quedarían para cada proceso como:
Asignaciones: $128^2 * 16 + 128 * 16 = 262144 + 2048 = 264192$.
Sumas: $128^2 * 16 = 262144$.
Productos: $128^2 * 16 = 262144$

c) **Si los procesadores P1 a P7 son iguales, y sus tiempos de asignación son 1, de suma 2 y de producto 3, y si P8 es 4 veces más lento, ¿Cuánto tarda el proceso total concurrente? ¿Cuál es el valor del speedup (Tiempo secuencial/Tiempo paralelo)?. Modifique el código para lograr un mejor speedup.**

p1 a p8 tienen igual número de operaciones.
Asignaciones umas = 262144 y Productos = 262144

unidades de tiempo de p1 a p7 (Asignaciones =1t, Sumas =2t y Productos =3t)
Total de tiempo por proceso = (264192 * 1)+(262144 * 2) +(262144 *3) =1574912
ero P8 es 4 veces más lento
total de tiempo de P8 =(1574912 * 4) = 6299648

T*iempo de ejecución total paralelo* *es 6299648*.

Usando los cálculos de a) obtenemos *el tiempo secuencial* con los tiempos de las operaciones del procesador más eficiente, es decir p1 a p7
Asignaciones: $128^3 + 128^2 = 2113536$
Sumas: $128^3 * 2 = 2097152 * 2 = 4194304$
Productos: $128^3 * 3 = 2097152 * 3 = 6291456$

Entonces el *tiempo que requiere la ejecución secuencial* es de 12599296 unidades de tiempo.

Speedup de 12599296/6299648 = 2

Esto puede ser mejorado si realizamos un mejor balance de carga haciendo que P8 trabaje sobre un strip más pequeño. Dándole a P8 solo dos filas y los demás procesando 18 filas obtenemos los siguientes cálculos:

p1 a p7
Asignaciones: $128^2 * 18 + 128 * 18 = 294912 + 2304 = 297216$.
Sumas: $(128^2 * 18) * 2 = 294912 * 2 = 589824$.
Productos: $(128^2 * 18) * 3 = 294912 * 3 = 884736$.
Entonces, P1…P7 ejecutan en 1771776 unidades de tiempo.

p8
Asignaciones: $128^2 * 2 + 128 * 2 = 32768 + 256 = 33024$.
Sumas: $(128^2 * 2) * 2 = 32768 * 2 = 65536$.
Productos: $(128^2 * 2) * 3 = 32768 * 3 = 98304$.
P8 consume $196864 * 4 = 787456$.

Tiempo paralelo de ejecución pasa a ser 1771776.

Speedup = 12599296/1771776 =7,1 logrando una gran mejora con respecto al algoritmo secuencial gracias al balance de carga realizado para evitar retrasos por parte de P8.

Dado el siguiente programa concurrente con memoria compartida:
X:=4; y:=2; z:=3;
Co
x:=x-z//z:=z*2//y:=z+4
Oc

a) **Cuáles de las asignaciones dentro de la sentencia co cumplen la propiedad de ASV. Justifique claramente.**

Una sentencia de asignación x = e satisface la propiedad de A lo sumo una vez sí:
(1) e contiene a lo sumo una referencia crítica y x no es referenciada por otro proceso, o
(2) e no contiene referencias críticas, en cuyo caso x puede ser leída por otro proceso

- X:=X-Z Cumple la propiedad de a lo sumo una vez ya que posee una única referencia crítica (Z) y X no es referenciada por otro proceso.
- Z:=Z*2 Cumple la propiedad de a lo sumo una vez porque Z no es una referencias críticas por lo tanto, Z puede ser leída por otro proceso.
- Y:=Z+4 Cumple la propiedad de a lo sumo una vez ya que posee una referencia crítica (Z) e Y no es referenciada por ningún otro proceso.

b) **Indique los resultados posibles de la ejecución. Justifique.**

Cada tarea se ejecuta sin ninguna interrupción hasta que termina (Si una sentencia no es atómica se puede cortar su ejecución pero al cumplir ASV la ejecución no se ve afectada) y las llamamos T1, T2 y T3 respectivamente obtenemos el siguiente subconjunto de historias:
T1, T2, T3 => X=1, Z=6, y=10
T1, T3, T2 => X=1, Z=6, y=7
T2, T1, T3 => X=-2, Z=6, y=10
T2, T3, T1 => X=-2, Z=6, y=10
T3, T1, T2 => X=1, Z=6, y=7
T3, T2, T1 => X=-2, Z=6, y=7
El valor de Z es siempre el mismo ya que no posee ninguna referencia crítica. Los valores de X e Y se ven afectados por la ejecución de T2 ya que sus resultados dependen de la referencia que hacen a la variable Z que es modificada. Entonces, si T1 y T3 se ejecutan antes que T2 ambas usarán el valor inicial de Z que es 3 obteniendo los resultados X=1 e Y=7; ahora si T2 se ejecuta antes que las demás los resultados serán X=-2 e Y=10 y por último, tenemos los casos en que T2 se ejecuta en medio con T1 antes y T3 después o con T3 antes y T1 después.

Nota 1: las instrucciones NO SON atómicas.
Nota 2: no es necesario que liste TODOS los resultados.

Dado el siguiente programa concurrente con memoria compartida:
x = 3; y = 2; z = 5;
co
x = y * z // z = z * 2 // y = y + 2x
oc

a) **Cuáles de las asignaciones dentro de la sentencia co cumplen la propiedad de "A lo sumo una vez". Justifique claramente.**

Siendo:
A: x = y * z
B: z = z * 2
C: y = y + 2x

A tiene 2 referencias críticas (a y, a z), por lo tanto no cumple ASV. (además x es leída en C).
B no tiene referencia crítica y es leída por otro (en A se lee z), por lo tanto cumple ASV.
C tiene 1 referencia crítica (a x) y además es leída por otro proceso (en A se lee y), por lo tanto no cumple ASV.

b) **Indique los resultados posibles de la ejecución. Justifique.**

Si se ejecutan en el orden A, B y C o A, C y B -> x = 10; z = 10; y = 22
Si se ejecutan en el orden C, B y A o B, C y A -> x = 80; z = 10; y = 8
Si se ejecutan en el orden C, A y B -> x = 40; z = 10; y = 8
Si se ejecutan en el orden B, A y C-> x = 20; z = 10; y = 42
Si se empieza a ejecutar A leyendo a y = 2, y en ese momento se ejecuta C leyendo a x = 3 (porque no terminó la asignación de A), y luego termina lo que falta de A y se ejecuta B: x = 10; z = 10; y = 8
Si se empieza a ejecutar A leyendo a y = 2, y en ese momento se ejecuta C leyendo a x = 3 (porque no terminó la asignación de A), y luego se ejecuta B y lo que falta de A: x = 20; z = 10; y = 8

Sea la siguiente solución propuesta al problema de alocación SJN (short Job Next):

```
Monitor SJN {
  Bool libre=true;
  Cond turno;
  Procedure request {
        If (not libre) wait (turno, tiempo);
        Libre = false;}
  Procedure release {
        Libre = true;
        Signal (turno); }
}
```

a) **¿Funciona correctamente con disciplina de señalización Signal and continue? Justifique.**

Con S&C un proceso que es despertado para poder seguir ejecutando es pasado a la cola de ready en cuyo caso su orden de ejecución depende de la política que se utilice para ordenar los procesos en dicha cola. Puede ser que sea retrasado en esa cola permitiendo que otro proceso ejecute en el monitor antes que el por lo que podría no cumplirse el objetivo del SJN.

b) **¿Funciona correctamente con disciplina de señalización signal and wait? Justifique.**

En cambio, con S&W se asegura que el proceso despertado es el próximo en ejecutar después de que el señalador ejecuta signal. Por lo tanto, SJN funcionaria correctamente de esta forma evitando que cualquier otro proceso listo para ejecutar le robe el acceso al proceso despertado.

Ø Definiciones:

Signal and Continue -> el proceso que hace el signal continua usando el monitor, y el proceso despertado pasa a competir por acceder nuevamente al monitor para continuar con su ejecución (en la instrucción que lógicamente le sigue al wait).

Ø *Signal and Wait* -> el proceso que hace el signal pasa a competir por acceder nuevamente al monitor, mientras que el proceso despertado pasa a ejecutar dentro del monitor a partir de instrucción que lógicamente le sigue al wait.

Utilice la técnica de "passing the condition" para implementar un semáforo fair usando monitores.

```
monitor Semaforo
{  int s = 1, espera = 0;  cond pos;

   procedure  P ()
      { if (s == 0) { espera ++; wait(pos);}
        else  s = s-1;
      };

   procedure V ()
      { if (espera == 0 ) s = s+1
        else { espera --; signal(pos);}
      };
};
```

Resuelva con monitores el siguiente problema:

Tres clases de procesos comparten el acceso a una lista enlazada: searchers, inserters y deleters. Los *searchers* sólo examinan la lista, y por lo tanto pueden ejecutar concurrentemente unos con otros. Los *inserters* agregan nuevos items al final de la lista; las inserciones deben ser mutuamente exclusivas para evitar insertar dos items casi al mismo tiempo. Sin embargo un insert puede hacerse en paralelo con uno o más searchers. Por último, los *deleters* remueven items de cualquier lugar de la lista. A lo sumo un deleter puede acceder la lista a la vez, y el borrado también debe ser mutuamente exclusivo con searchers e inserciones.

```
monitor Controlador_ListaEnlazada{
int numSearchers=0,  numInserters=0,  numDeleters =0;
cond searchers,  inserters, deleters;

procedure pedir_Deleter(){
     while (numSearchers>0 OR numInserters>0 OR  numDeleters >0){ wait(deleters);}
     numDeleters=numDeleters+1;
}

procedure liberar_Deleter(){
    numDeleters=numDeleters-1;
    signal(inserters);
    signal(deleters);
    signal_all(searchers);
}

procedure pedir_Searcher(){
    while(numDeleters >0){ wait(searchers);}
```

```
    numSearchers=numSearchers+1;
}

procedure liberar_Searcher(){
   numSearchers=numSearchers-1;
   if(numSearchers==0 AND numInserters==0  ){
       signal(deleters);
   }
}

procedure pedir_Inserter(){
    while(numDeleters >0 OR numInserters>0 ){ wait(inserters);}
    numInserters=numInserters+1;
}

procedure liberar_Inserter(){
   numInserters=numInserters-1;
  signal(inserters);
   if(numSearchers==0  ){
       signal(deleters);
   }
}
}
```

```
process Searchers[i=1..S] {
   Controlador_ListaEnlazada.pedir_Searcher();
   <Realiza búsqueda en la lista>
   Controlador_ListaEnlazada.liberar_Searcher();
}

process Inserters[j=1..I] {
   Controlador_ListaEnlazada.pedir_Inserter();
   <Inserta en la lista>
   Controlador_ListaEnlazada.liberar_Inserter();
}

process Deleters[k=1..D] {
   Controlador_ListaEnlazada.pedir_Deleter();
   <Borra en la lista>
   Controlador_ListaEnlazada.liberar_Deleter();
```

En los protocolos de acceso a sección crítica vistos en clase, cada proceso ejecuta el mismo algoritmo. Una manera alternativa de resolver el problema es usando un proceso coordinador. En este caso, cuando cada proceso SC[i] quiere entrar a su sección crítica le avisa al coordinador, y espera a que éste le de permiso. Al terminar de ejecutar su sección crítica, el proceso SC[i] le avisa al coordinador.
Desarrolle protocolos para los procesos SC[i] y el coordinador usando sólo variables compartidas (no tenga en cuenta la propiedad de eventual entrada).

```
int aviso [1:n]=([n]0), permiso [1:n]=([n]0);

process SC[i = 1 to N] {
SNC;
permiso[i] = 1; # Protocolo
while (aviso[i]==0) skip; # de entrada
SC;
```

```
aviso[i] = 0; # Protocolo de salida
SNC;
}

process Coordinador {
int i = 1;
while (true) {
        while (permiso[i]==0) i = i mod N +1;
        permiso[i] = 0;
        aviso[i] = 1;
        while (aviso[i]==1) skip;
}
}
```

8. Describa la solución utilizando la criba de Eratóstenes al problema de hallar los primos entre 2 y n. Cómo termina el algoritmo? Qué modificaría para que no termine de esa manera?

La criba de Eratóstenes es un algoritmo clásico para determinar cuáles números en un rango son primos. Supongamos que queremos generar todos los primos entre 2 y n. Primero, escribimos una lista con todos los números:

2 3 4 5 6 7 ... n

Comenzando con el primer número no tachado en la lista, 2, recorremos la lista y borramos los múltiplos de ese número. Si n es impar, obtenemos la lista:

2 3 5 7 ... n

En este momento, los números borrados no son primos; los números que quedan todavía son candidatos a ser primos. Pasamos al próximo número, 3, y repetimos el anterior proceso borrando los múltiplos de 3. Si seguimos este proceso hasta que todo número fue considerado, los números que quedan en la lista final serán todos los primos entre 2 y n.

Para solucionar este problema de forma paralela podemos emplear un pipeline de procesos filtro. Cada filtro recibe una serie de números de su predecesor y envía una serie de números a su sucesor. El primer número que recibe un filtro es el próximo primo más grande; le pasa a su sucesor todos los números que no son múltiplos del primero.

El siguiente es el algoritmo pipeline para la generación de números primos. Por cada canal, el primer número es primo y todos los otros números no son múltiplo de ningún primo menor que el primer número:

```
Process Criba[1]
{   int p = 2;

    for [i = 3 to n by 2]  Criba[2] ! (i);
}

Process Criba[i = 2 to L]
{   int p, proximo;

    Criba[i-1] ? P;
    do Criba[i-1] ? (Proximo) →
            if  ((proximo MOD p) <> 0 ) →  Criba[i+1] ! (proximo); fi
    od
}
```

El primer proceso, Criba[1], envía todos los números impares desde 3 a n a Criba[2]. Cada uno de los otros procesos recibe una serie de números de su predecesor. El primer número p que recibe el proceso Criba[i] es el i-ésimo primo. Cada Criba[i] subsecuentemente pasa todos los otros números que recibe que no son múltiplos de su primo p. El número total L de procesos Cribe debe ser lo suficientemente grande para garantizar que todos los primos hasta n son generados. Por ejemplo, hay 25 primos menores que 100; el porcentaje decrece para valores crecientes de n.

El programa anterior termina en deadlock, ya que no hay forma de saber cuál es el último número de la secuencia y cada proceso queda esperando un próximo número que no llega. Podemos fácilmente modificarlo para que termine normalmente usando centinelas, es decir que al final de los streams de entrada son marcados por un centinela

EOS (End Of Stream).

```
Process Criba[1] {
    INT p=2;
    for [i = 3 to n by 2] Criba[2] ! i # pasa impares a Criba[2]
    Criba[2] ! –1;
}

Process Criba[i = 2 TO L] {
    INT p, proximo;
    boolean seguir = true;
    Criba[i-1] ? p                  # p es primo
    do (seguir); Criba[i-1] ? proximo -> # recibe próximo candidato
        if (proximo = -1) {
                seguir = false;
                Criba[i+1] ! -1;
        }
        else if ((proximo MOD p) <> 0)   # si es primo
                Criba[i+1] ! proximo;      # entonces lo pasa
        fi
    od
}
```

Suponga los siguientes programas concurrentes. Asuma que "función" existe, y que los procesos son iniciados desde el programa principal.

P1	chan canal (double)
	process grano1 {
	int veces, i;
	double sum;
	for [veces= 1 to 10] {
	for [i = 1 to 10000]
	
	sum=sum+funcion(i);
	send canal (sum); } }
		process grano2 {
		int veces; double sum;
		for [veces= 1 to 10] {
		receive canal (sum);
		printf (sum);
		}
		}
			P2	chan canal (double)
				process grano1 {
				int veces, i;
				double sum;
				for [veces= 1 to 10000] {
				for [i = 1 to 10]
				
				sum=sum+i;
				send canal (sum); } }
					process grano2 {
					int veces; double sum;
					for [veces= 1 to 10000] {
					receive canal (sum);
					printf (sum);
					}
					}

a) Analice desde el punto de vista del número de mensajes.
En la P1, sólo se envían 10 mensajes al proceso grano2. En P2 se envían 10000 mensajes.
b) Analice desde el punto de vista de la granularidad de los procesos.
En P1, el envío de los mensajes se realiza después de largos períodos de ejecución ya que entre cada send se ejecuta una iteración de 10000 unidades de tiempo, esto nos asegura que la comunicación entre los dos procesos es poco frecuente. Dadas dichas características podemos decir, que desde el punto de vista de la granularidad, P1 es de granularidad gruesa ya que la comunicación ente los procesos no es de manera reiterada. Al tener mayor granularidad disminuye la concurrencia y la sobrecarga de bloqueos.
En P2, el envío de mensajes se realiza en intervalos cortos de tiempo (entre la ejecución de cada send sólo se ejecuta un for de 1 a 10), aumentando considerablemente la comunicación respecto de P1. Por lo tanto, podemos decir que P2 es de granularidad fina, ya que en cada iteración el volumen de comunicación aumenta, por lo tanto la relación cómputo / comunicación disminuye. Al disminuir la granularidad aumenta la concurrencia pero también aumenta la sobrecarga de bloqueos.
c) Cuál de los programas le parece más adecuado para ejecutar sobre una arquitectura de tipo cluster de PCs? Justifique.
La implementación más adecuada para este tipo de arquitecturas es P1, por ser de granularidad gruesa. Al tratarse de una arquitectura con memoria distribuida la

comunicación entre los procesos es más costosa ya que cada proceso puede ejecutarse en computadores diferentes, por lo tanto sería más eficiente que la sobrecarga de comunicación sea lo más baja posible, y dicha característica la brinda la granularidad gruesa.

Suponga los siguientes programas concurrentes. Asuma que EOS es un valor especial que indica el fin de la secuencia de mensajes, y que los procesos son iniciados desde el programa principal.

P1

```
chan canal (double)
process Genera {
  int fila, col;   double sum;
  for [fila= 1 to 10000]
    for [col = 1 to 10000]
       send canal (a(fila,col));
  send canal (EOS)   }
```

```
process Acumula {
  double valor, sumT;
  sumT=0;
  receive canal (valor);
  while valor<>EOS {
     sumT = sumT + valor
     receive canal (valor); }
  printf (sumT);
}
```

P2

```
chan canal (double)
process Genera {
  int fila, col;   double sum;
  for [fila= 1 to 10000] {
    sum=0;
    for [col = 1 to 10000]
         sum=sum+a(fila,col);
    send canal (sum);   }
  send canal (EOS)   }
```

```
process Acumula {
  double valor, sumT;
  sumT=0;
  receive canal (valor);
  while valor<>EOS {
     sumT = sumT + valor
     receive canal (valor); }
  printf (sumT);
}
```

a) ¿Qué hacen los programas?

Ambos programas realizan la suma de todos los elementos de una matriz pero difieren en la forma de llegar al resultado. P1 Genera le envía los 10000^2 valores a acumula para que este los sume mientras que en P2 Genera solo le envía 10000 valores ya que el mismo se encarga de sumar toda una fila. Es decir, le envía a acumula la suma de todos los elemento de cada fila.

b) Analice desde el punto de vista del número de mensajes.

P1 realiza 10000^2 envíos de mensajes (uno por cada elemento de la matriz) mientras que P2 solo realiza 10000 (uno por cada fila de la matriz).

c) Analice desde el punto de vista de la granularidad de los procesos.

Se puede decir que el proceso P2 es de grano más grueso que el proceso P1 ya que realiza más procesamiento en Genera y esto hace que se necesite un menor número de comunicaciones con Acumula para llegar al resultado.

d) ¿Cuál de los programas le parece más adecuado para ejecutar sobre una arquitectura de tipo cluster de PCs? Justifique.

Las arquitecturas del tipo cluster pueden considerarse arquitecturas de grano grueso ya que poseen muchos procesadores con mucha capacidad de cómputo pero con poca capacidad para la comunicación haciéndolos más apropiados para ejecutar programas de grano grueso que requieren de menos comunicación que capacidad de computo. Por lo tanto, P2 sería más apropiado para ejecutarse sobre este tipo de arquitecturas.

Dada la siguiente solución con monitores al problema de alocación de un recurso con múltiples unidades, transforme la misma en una solución utilizando mensajes asincrónicos.

```
Monitor Alocador_Recurso {
   INT disponible = MAXUNIDADES;
   SET unidades = valores iniciales;
   COND libre;              # TRUE cuando hay recursos
   procedure adquirir( INT  Id )  {
        if (disponible == 0)  wait(libre)
                else  disponible = disponible - 1; remove(unidades, id);      }
   procedure liberar( INT id ) {
        insert(unidades, id);
        if (empty(libre))  disponible := disponible + 1
        else  signal(libre);            }
}
```

```
type clase_op = enum(adquirir, liberar);
chan request(int idCliente, claseOp oper, int idUnidad );
chan respuesta[n] (int id_unidad);

Process Administrador_Recurso
{  int disponible = MAXUNIDADES;
   set unidades = valor inicial disponible;
   queue pendientes;
   while (true)
     {  receive request (IdCliente, oper, id_unidad);
        if (oper == adquirir)
           {   if (disponible > 0)
                 {  disponible = disponible - 1;
                    remove (unidades, id_unidad);
                    send respuesta[IdCliente] (id_unidad);
                 }
               else  insert (pendientes, IdCliente);
           }
        else
           { if empty (pendientes)
                 {  disponible= disponible + 1;
                    insert(unidades, id_unidad);
                 }
             else
                 {  remove(pendientes, IdCliente);
                    send respuesta[IdCliente](id_unidad);
                 }
           }
     } //while
}   //process  Administrador_Recurso

Process Cliente[i = 1 to n]
{   int id_unidad;
    send request(i, adquirir, 0);
    receive respuesta[i](id_unidad);
     //Usa la unidad
    send request(i, liberar, id_unidad);
}
```

Dados los siguientes dos segmentos de código, indicar para cada uno de los ítems si son equivalentes o no. Justificar cada caso (de ser necesario dar ejemplos).

Segmento 1

```
...
int cant=1000;
DO   (cant < -10);  datos?(cant) → Sentencias1
   □ (cant > 10);  datos?(cant) → Sentencias2
   □ (INCOGNITA);  datos?(cant) → Sentencias3
END DO
...
```

Segmento 2

```
...
int cant=1000;
While (true)
{ IF   (cant < -10);  datos?(cant) → Sentencias1
     □ (cant > 10);  datos?(cant) →Sentencias2
     □ (INCOGNITA);  datos?(cant) → Sentencias3
   END IF
}
...
```

En todos los casos, en primera instancia el valor de cant va a ser 1000, por lo tanto entra por la guarda (cant>10). Si en esa guarda el valor de cant se modifica (porque llega por el canal, o porque lo modifica Sentencia 2), las guardas deben cubrir todos los posibles valores que toma la variable, para que ambos segmentos sean equivalentes. De no ser así, cuando en el segmento 1 todas las guardas son falsas, el do termina y sigue la ejecución del programa, mientras que el segmento 2, se queda en un bucle hasta que la variable sea modificada por otro proceso.

a) **INCOGNITA equivale a: (cant = 0),**

No son equivalentes ya que el DO en segmento 1 terminaría para valores de cant (1...10) (-1…-10) mientras que en segmento 2 el if fallaría pero la iteración se seguiría realizando.

b) **INCOGNITA equivale a: (cant > -100)**

Son equivalentes con estas condiciones no hay ningún caso en el que todas las guardas fallen por lo tanto el DO no termina nunca, obteniendo el mismo comportamiento que el while true con el IF.

c) **INCOGNITA equivale a: ((cant > 0) or (cant < 0))**

No son equivalentes ya que el DO termina si cant=0 porque todas las guardas son false.

d) **INCOGNITA equivale a: ((cant > -10) or (cant < 10))**

No son equivalente ya que para cant = 10 o cant=-10 segmento 1 terminaría su ejecución del DO porque todas las guardas serían falsas.

e) **INCOGNITA equivale a: ((cant >= -10) or (cant <= 10))**

Serian equivalentes ya que se estarían eliminando en segmento 1 los valores puestos en d) que provocarían que el DO terminase.

Sea "ocupados" una variable entera inicializada en N que representa la cantidad de slots ocupados de un buffer, y sean P1 y P2 dos programas que se ejecutan de manera concurrente, donde cada una de las instrucciones que los componen son atómicas.

```
P1::
    if (ocupados < N) then
     begin
        buffer := elemento_a_agregar;
        ocupados := ocupados + 1;
     end;
```

```
P2::
    if (ocupados > 0) then
  begin
      ocupados := ocupados - 1;
      elemento_a_sacar:= buffer;
  end;
```

El programa funciona correctamente para asegurar el manejo del buffer? Si su respuesta es afirmativa justifique. Sino, encuentre una secuencia de ejecución que lo verifique y escríbala, y además modifique la solución para que funcione correctamente (Suponga buffer, elemento_a_agregar y elemento_a_sacar variables declaradas).

El programa no funciona correctamente, observemos la siguiente situación para comprobarlo:

El buffer está lleno. P2 ejecuta el if(ocupados > 0) que resulta true, luego decrementa la variable ocupados. En ese momento el P1 ejecuta su respectivo if, que en ese momento es verdadero, y procede a agregar al buffer un nuevo elemento descartando al elemento que P2 todavía no saco del buffer.

Esta situación es fácil de solucionar, simplemente se posterga el decremento de ocupados de P2 hasta después de obtener el elemento del buffer.

```
P2::
If(ocupado>0)then
Begin
    Elemento_a_sacar:=buffer;
    Ocupado:=ocupado-1;
End;
```

Sea "cantidad" una variable entera inicializada en 0 que representa la cantidad de elementos de un buffer, y sean P1 y P2 dos programas que se ejecutan de manera concurrente, donde cada una de las instrucciones que los componen son atómicas.

```
P1::
    if (cantidad = 0) then
     begin
        cantidad:= cantidad + 1;
        buffer := elemento_a_agregar;
     end;
```

```
P2::
    if (cantidad > 0) then
    begin
        elemento_a_sacar:= buffer;
        cantidad:= cantidad - 1;
    end;
```

Además existen dos alumnos de concurrente que analizan el programa y opinan lo siguiente:
"Pepe: este programa funciona correctamente ya que las instrucciones son atómicas".
"José: no Pepe estás equivocado, hay por lo menos una secuencia de ejecución en la cual funciona erróneamente"
¿Con cuál de los dos alumnos está de acuerdo? Si está de acuerdo con Pepe justifique su respuesta. Si está de acuerdo con José encuentre una secuencia de ejecución que verifique lo que José opina y escríbala, y modifique la solución para que funcione correctamente (Suponga buffer y elemento variables declaradas). (22-04-2009)

Estoy de acuerdo con José. Ya que existen secuencias de ejecución en las cuales no funciona correctamente.

Por ejemplo si P1 ejecuta su primera linea y suma cantidad:=cantidad + 1, y corta su ejecución, se ejecuta P2, la condición de cantidad>0 da verdadera pero no existe nada en el buffer y P2 en su primera linea ejecuta elemento_a_sacar := buffer.

4. Dado el siguiente bloque de código, indique para cada inciso que valor queda en aux, o si el código queda bloqueado. Justifique sus respuestas.

```
aux = -1;
...
if (A == 0); P2?(aux) -> aux = aux + 2;
□ (A == 1); P3?(aux) -> aux = aux + 5;
□ (B == 0); P3?(aux) -> aux = aux + 7;
end if;
...
```

a) Si el valor de A=1 y B=2 antes del IF, y sólo P2 envía el valor 6.
b) Si el valor de A=0 y B=2 antes del IF, y sólo P2 envía el valor 8.
c) Si el valor de A=2 y B=0 antes del IF, y sólo P3 envía el valor 6.
d) Si el valor de A=2 y B=1 antes del IF, y sólo P3 envía el valor 9.
e) Si el valor de A=1 y B=0 antes del IF, y sólo P3 envía el valor 14.
f) Si el valor de A=0 y B=0 antes del IF, y P3 envía el valor 9 y P2 el valor 5.

i. Si el valor de A = 1 y B = 2 antes del if, y solo P2 envia el valor 6.

Se queda bloquado en la guarda A==1 ya que la condicion es la unica verdadera pero P3 no se ejecuto.

ii. Si el valor de A = 0 y B = 2 antes del if, y solo P2 envia el valor 8.

Entra en la primer guarda (unica con condicion verdadera) y se ejecuta P2. Luego el valor de aux sera 10.

iii. Si el valor de A = 2 y B = 0 antes del if, y solo P3 envia el valor 6.

Entra en la tercer guarda (unica con condicion verdadera) y se ejecuta P3. Luego el valor de aux sera 13.

iv. Si el valor de A = 2 y B = 1 antes del if, y solo P3 envia el valor 9.

El if no tiene efecto ya que ninguna condicion es verdadera. Aux mantiene el valor (-1).

v. Si el valor de A = 1 y B = 0 antes del if, y solo P3 envia el valor 14.

Puede entrar en la segunda o tercer guarda (eleccion no determinista) ya que ambas son verdaderas y la sentencia de comunicación puede ejecutarse inmediatamente (no hay bloqueo).

Caso A==1: el valor de aux sera 19.

Caso B==0: el valor de aux sera 21.

vi. Si el valor de A = 0 y B = 0 antes del if, P3 envia el valor 9 y P2 el valor 5.

Tanto la condicion de la primer guarda y la condicion de la ultima son verdaderas; ademas P3 y P2 se ejecutaron, por lo tanto no hay bloqueo. Como la eleccion es no determinista, pueden suceder dos casos:

Caso A==0: el valor de aux sera 7.

Caso B==0: el valor de aux sera 16.

2) Dado el siguiente programa concurrente con memoria compartida, y suponiendo que todas las variables están inicializadas en 0 al empezar el programa y las instrucciones NO son atómicas. Para cada una de las opciónes indique verdadero o falso. En caso de ser verdadero indique el camino de ejecución para llegar a ese valor, y en caso de ser falso justifique claramente su respuesta.

P1::	P2::	P3::

if(x = 0) then
y:= 4*x + 2;
x:= y + 2 + x;
	if (x >= 0) then
	x:= x + 1;
		x:= x*8 + x*2 + 1;

a) El valor de x al terminar el programa es 9.

b) El valor de x al terminar el programa es 6.

c) El valor de x al terminar el programa es 11.

d) Y siempre termina con alguno de los siguientes valores: 10 o 6 o 2 o 0.

a) Verdadero: p2->p1(solo if)->p3->p1(asignación de x e y)

b) Verdadero, caso1: p1(solo if y asignación de y) ->p3->p2->p1(resto)
caso 2: p1(solo if y asignación de y) ->p3->p1(resto) ->p2

c) Verdadero

d) Verdadero.

Y=0: El único caso en el que Y sería 0, es que se ejecute P3 primero alterando el valor de X por lo que al ejecutarse P1 no realizaría ningún calculo.

Y=2: Siempre que asignación se realice antes de que los demás procesos alteren X (X=0). Si primero se ejecuta P1 completando la asignación de Y, el valor final de Y es 2 ya que ningún proceso vuelve a modificarla o modifico el valor de X.

Y=6: Si primero se ejecuta P2 terminando su ejecución porque x=0 y luego P1 evalúa el if y el control se pasa a P3 en ese instante, X=1 y al volver a realizar la asignación de Y en P1 el valor de Y=6.

Y=10: Se ejecuta P3 habilitando la asignación de P2 por lo que el valor de X=2 y luego se realiza la asignación de Y.

Sea el problema de ordenar de menor a mayor un arreglo de A[1..n]

1. Escriba un programa donde dos procesos (cada uno con n/2 valores) realicen la operación en paralelo mediante una serie de intercambios.

```
Process P1
{  int nuevo, a1[1:n/2]; const mayor = n/2;
   ordenar a1 en orden no decreciente
   P2 ! (a1[mayor]);
   P2 ? (nuevo)
   do  a1[mayor] > nuevo →
       poner nuevo en el lugar correcto en a1, descartando el viejo a1[mayor]
       P2 ! (a1[mayor]);
       P2 ? (nuevo);
    od
}

Process P2
{  int nuevo, a2[1:n/2]; const menor = 1;
   ordenar a2 en orden no decreciente
   P1 ? (nuevo);
   P1 ! (a2[menor]);
   do  a2[menor] < nuevo →
       poner nuevo en el lugar correcto en a2, descartando el viejo a2[menor]
       P1 ? (nuevo);
       P1 ? (a2[menor]);
    od
}
```

2. ¿Cuántos mensajes intercambian en el mejor de los casos? ¿Y en el peor de los casos?

En el mejor caso, los procesos en el programa necesitan intercambiar solo un par de valores. Solo intercambia dos valores, para comprobar que ya están ordenados. Esto ocurrirá si los n/2 valores menores están inicialmente en P1, y los n/2 mayores en P2. Así que solo requieren 2 mensajes, uno por cada valor intercambiado.
En el peor caso (que ocurrirá si todo valor está inicialmente en el proceso equivocado) los procesos tendrán que intercambiar n/2 valores para tener cada valor en el proceso correcto. Esto equivale a n mensajes, uno por cada valor intercambiado (n/2 valores de p1 a p2 y n/2 valores de p2 a p1 = (n/2)*2=n)

a. 3. Utilice la idea de 1), extienda la solución a K procesos, con n/k valores c/u (“odd-even-exchange sort”).

Asumimos que existen n procesos P[1:n] y que n es par. Cada proceso ejecuta una serie de rondas. En las rondas impares, los procesos impares P[odd] intercambian valores con el siguiente proceso impar P[odd+1] si el valor esta fuera de orden. En rondas pares, los procesos pares P[even] intercambia valores con el siguiente proceso par P[even+1] si los valores estan fuera de orden. P[1] y P[n] no hacen nada en las rondas pares.

```
//k es el numero de procesos
// n cantidad de nueros a ordenar

process Proc[i:1..k] {
        int largest = n/k;
        int smallest = 1;
        int a[1..k];
        int dato;

        //se ordena el arreglo a de menor a mayor

   for(ronda=1;ronda<=k;ronda++) {

                # si el proceso tiene = paridad que la ronda, pasa valores para adelante
                if(i mod 2 == ronda mod 2) {ronda par y proceso par o ronda impar y
proceso impar
                        if(i!=k) {
                                proc[i+1]!a[largest];
                                proc[i+1]?dato;
                                while(a[largest]>dato) {

                                  //inserto dato en a ordenado, pisando a[largest]

                                  proc[i+1]!a[largest];
                                  proc[i+1]?dato;
                                }
```

```
            }
        }
        else{
            if(i!=1) { # si tiene distinta paridad, recibe valores desde atrás

                        proc[i-1]?dato;
                        proc[i-1]!a[smallest];
                        while(a[smallest]<dato) {

                                //inserto dato en a ordenado, pisando a[smallest]

                                proc[i-1]?dato;
                                proc[i-1]!a[smallest];
                        }
            }
        }
    }
}
```

b. ¿Cuántos mensajes intercambian en 3) en el mejor caso? ¿Y en el peor de los casos?

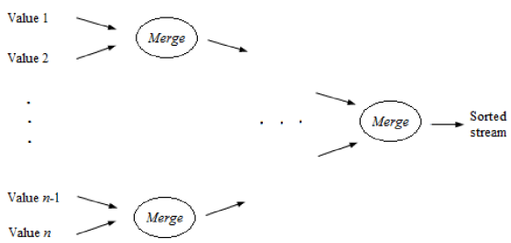
Si cada proceso ejecuta suficientes rondas para garantizar que la lista estará ordenada (en general, al menos k rondas), en el k-proceso, cada uno intercambia hasta (n/k)+1 mensajes por ronda. El algoritmo requiere hasta $k^2 * (n/k+1)$.
Se puede usar un proceso coordinador al cual todos los procesos le envían en cada ronda si realizaron algún cambio o no. Si al recibir todos los mensajes el coordinador detecta que ninguno cambio nada les comunica que terminaron. Esto agrega overhead de mensajes ya que se envían mensajes al coordinador y desde el coordinador. Con n procesos tenemos un overhead de 2*k mensajes en cada ronda.

Nota: Utilice un mecanismo de pasaje de mensajes, justifique la elección del mismo.

PMS es más adecuado en este caso porque los procesos deben sincronizar de a pares en cada ronda por lo que PMA no sería tan útil para la resolución de este problema ya que se necesitaría implementar una barrera simétrica para sincronizar los procesos de cada etapa.

Suponga los siguientes métodos de ordenación de menor a mayor para n valores (n par y potencia de 2), utilizando pasaje de mensajes:

1. **Un pipeline de filtros. El primero hace input de los valores de a uno por vez, mantiene el mínimo y le pasa los otros al siguiente. Cada filtro hace lo mismo: recibe un stream de valores desde el predecesor, mantiene el más chico y pasa los otros al sucesor.**
2. **Una red de procesos filtro (como la de la figura).**



3. **Odd/even exchange sort. Hay n procesos P[1:n], Cada uno ejecuta una serie de rondas. En las rondas "impares", los procesos con número impar P[impar] intercambian valores con P[impar+1]. En las rondas "pares", los procesos con número par P[par] intercambian valores con P[par+1] (P[1] y P[n] no hacen nada en las rondas "pares"). En cada caso, si los números están desordenados actualizan su valor con el recibido.**

Asuma que cada proceso tiene almacenamiento local sólo para dos valores (el

próximo y el mantenido hasta ese momento).

a) **¿Cuántos procesos son necesarios en 1 y 2? Justifique.**

Para la alternativa del pipeline de filtros se necesitan n procesos ya que cada uno de ellos calculara un mínimo y pasara el resto de los valores. Por lo tanto, para terminar de procesar todos los números hacen falta tantos procesos como números se quieran ordenar.

Para una red de procesos filtro se necesitan n-1 procesos, ya que al ser un árbol binario con $\log_2$ n niveles tenemos 2^n -1 nodos o procesos.

b) **¿Cuántos mensajes envía cada algoritmo para ordenar los valores? Justifique.**

Pipeline: Asignándole a cada proceso un número entero consecutivo, siendo el primer proceso el 1, el segundo el 2, así sucesivamente hasta el último proceso n, tenemos que:
Al proceso 1 le llegan n mensajes, y envía n-1 mensajes, porque se queda con el valor más chico de todos los que le llegan y va enviando los demás.
Al proceso 2 le llegan n-1 mensajes y envía n-2 mensajes, por la misma razón del proceso anterior.
Y así sucesivamente, hasta el proceso n, el cual recibe 1 (n-(n-1)) mensaje y no manda ningún (n-n) mensajes.
En conclusión se envían (n(n+1))/2 mensajes
A este resultado se le pueden sumar los mensajes de EOS, cada proceso envía un EOS teniendo n mensajes EOS.

En la red de procesos filtro: siendo n la cantidad de números a ordenar y $\log_2$ n el total de niveles de la red tenemos que la cantidad de mensajes es (n * $\log_2$ n).
A lo que podemos sumarle nuevamente los mensajes de EOS, que son los n mensajes que llegan con los números a ordenar (primeros nodos) y el que envía cada nodo (n-1) es decir que tenemos (n-1) + n = 2n – 1 mensajes extra.

Odd/even Exchange sort : Si cada proceso ejecuta suficientes rondas para garantizar que la lista estará ordenada (en general, al menos k rondas), en el k-proceso, cada uno intercambia hasta (n/k+1) mensajes por ronda. El algoritmo requiere hasta k^2 * (n/k+1).

c) **¿En cada caso, cuáles mensajes pueden ser enviados en paralelo (asumiendo que existe el hardware apropiado) y cuáles son enviados secuencialmente? Justifique.**

En el pipeline de filtros en un instante determinado (pipeline lleno) se podrán enviar en paralelo tantos mensajes como procesos tenga el pipeline ya que como cada uno de los procesos recibe, procesa y envía pueden todos estarse enviando los valores en un momento determinado.

En una red de filtros cada proceso envía de a un mensaje por vez ya que el funcionamiento interno de cada uno es igual a la de los filtros en un pipeline pero la diferencia está en la distribución e interacción entre ellos. Por lo tanto, se pueden enviar en paralelo tantos mensajes como procesos haya en el nivel en el que se esté dentro de la red.

Por último, en odd/even Exchange sort pueden enviarse en paralelo tantos mensajes como procesos se encuentren involucrados en una ronda ya que todos enviaran un valor.

d) **¿Cuál es el tiempo total de ejecución de cada algoritmo? Asuma que cada operación de comparación o de envío de mensaje toma una unidad de tiempo. Justifique.**

Algoritmo i: Utilizando el cálculo de la cantidad de mensajes del punto a) y razonando que cada vez que se envía un mensaje se realiza una comparación antes, tenemos que el tiempo de ejecución será igual a la cantidad de mensajes multiplicada por 2 unidades de tiempo (1 comparación + 1 mensaje): 2*((n(n+1))/2) = 2 n (n + 1) = n (n + 1) => O(n2)
Algoritmo ii: Por la misma razón que el algoritmo anterior tenemos:
2 (n * $\log_2$ n) => O(n)
Algoritmo iii: Cada proceso en cada ronda hace una comparación, envía, recibe y hace una asignación, por eso tenemos que por cada proceso se tarda 4 unidades. Como se

necesitan n rondas en el peor de los casos, se realizará 4n unidades de tiempo en total, por lo tanto es de O(n).

“ESTE PUNTO MUY DUDOSO”

Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, el speedup (S) está regido por la función S=p-1 y en el otro por la función S=p/2. ¿Cuál de las dos soluciones se comportará más eficientemente al crecer la cantidad de procesadores? Justifique claramente.

El que mejor se comportara con mayor número de procesadores es aquel cuya función de speedup es p-1 por definición de speedup su rango esta entre 0 y p entonces con esta función el speedup es más cercano a p. Si comparamos las eficiencias tenemos que:
E=S/p entonces tenemos p-1/p y (p/2)/p cuanto más grande sea p mayor eficiencia tendrá la primer solución ya que su valor será casi uno y la otra solución no alcanzara nunca una mayor eficiencia que la mitad.

Ahora suponga S=1/p y S=1/p^2.
Es más eficiente la solución con speedup 1/p su speedup siempre será mayor. Además su eficiencia será siempre mayor que la de la segunda solución, ambas a medida que crecen los procesadores van disminuyendo su eficiencia pero la función 1/p decrece más lentamente.

Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, el speedup (S) está regido por la función S=p/3 y en el otro por la función S=p-3. ¿Cuál de las dos soluciones se comportará más eficientemente al crecer la cantidad de procesadores? Justifique claramente.

Suponiendo el uso de 5 procesadores:

Solución 1 => S=5/3=1,66
Solución 2 => S=5-3=2

Ahora, incrementamos la cantidad de procesadores suponemos 100 procesadores:

Solución 1 => S=100/3=33,33
Solución 2 => S=100-3=97

Podemos decir, que a medida que p tiende a infinito, para la solución 1 siempre el Speedup será la tercera parte en cambio para la solución 2 el valor "-3" se vuelve despreciable. Por lo tanto la solución 2 es la que se comporta más eficientemente al crecer la cantidad de procesadores.

Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, el speedup(s) esta regido por la función S= p-4 y el otro por la función S= p/3 para p > 4. ¿Cuál de las dos soluciones se comportara más eficientemente al crecer la cantidad de procesadores?

Si la cantidad de procesadores crece, la primera solución se comportara más eficientemente. Ya que si p es muy grande y se divide en 3 partes, estas no serán tan grandes. En cambio, si p es muy grande y se le resta 4, el resultado seguiría siendo muy grande.

Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, la eficiencia está regido por la función E=1/p y

en el otro por la función E= 1/p^2. ¿Cuál de las dos soluciones se comportará más eficientemente al crecer la cantidad de procesadores? Justifique.

Claramente se observa que a partir de p=2, E1=1/p será mayor que E2= 1/p^2, observemos los siguientes casos:
I. p = 1, tenemos E1 = 1/1 = 1 y E2 = 1/1 = 1
II. p = 2, tenemos E1 = 1/2 = 0,5 y E2 = 1/4 = 0,25
III. p = 3, tenemos E1 = 1/3 = 0,33 y E2 = 1/9 = 0,11
Como se puede apreciar, a partir de p=2 E1 se mantiene más eficiente que E2, pero sin embargo ambos van a decrecer conforme p crezca, por lo tanto ninguno de los dos se comportará más eficiente al crecer la cantidad de procesadores.

Suponga que el tiempo de ejecución de un algoritmo secuencial es de 1000 unidades de tiempo, de las cuales el 80% corresponden a código paralelizable. ¿Cuál es el límite en la mejora que puede obtenerse paralelizando el algoritmo?

El límite de mejora se da teniendo 800 procesadores los cuales ejecutan una unidad de tiempo obteniendo un tiempo paralelo de 201 unidades de tiempo. El speedup mide la mejora de tiempo obtenida con un algoritmo paralelo comparándola con el tiempo secuencial. S=Ts/Tp=1000/201= 4,97~5 aunque se utilicen más procesadores el mayor speedup alcanzable es el anterior cumpliéndose así la ley de Amdahl que dice que para un problema existe un límite en la paralización del mismo que no depende del número de procesadores sino que depende de la cantidad de código secuencial.

Si el tiempo de ejecución de un algoritmo secuencial es de 1000 unidades y el porcentaje de unidades paralelizables es del 80%, quiere decir que 200 unidades (el 20%) deben ejecutarse secuencialmente, por lo tanto el tiempo mínimo esperable es de 200 para un procesador. Por esta razón nos conviene utilizar una cantidad de procesadores tal que los mantenga a todos ocupados esa cantidad de tiempo, es decir en este caso conviene tener 5 procesadores porque si tuviéramos más un procesador estaría trabajando 200 unidades de tiempo y los demás terminarían de ejecutar las instrucciones antes, y tendrían que esperarlo.

Suponga que el tiempo de ejecución de un algoritmo secuencial es de 8000 unidades de tiempo, de las cuales solo el 90% corresponde a código paralelizable. ¿Cuál es el límite en la mejora que puede obtenerse paralelizando el algoritmo? Justifique.

Si yo tengo 8000 unidades de tiempo de las cuales el 90% de éstas pueden ser paralelizables entonces:

La mejora de la paralelizacion de un algoritmo se mide con el Speedup.

El tiempo total del algoritmo lo podemos dividir en:

T = Tsec + Tpar
T = 800 + 7200

Ahora suponiendo que el tiempo paralelizable (Tpar) que es 7200 podemos reducirlo a 1 si pudiéramos utilizar 7200 procesadores, y el tiempo secuencial (Tsec) que es 800 no puede disminuir de ese número porque se debe ejecutar solo por un procesador, por lo tanto:

Tmejor = 800 + 1 = 801

Y ahora calculamos el límite de mejora que podemos llegar a obtener:

S = T / Tmejor (paralelo)

S = 8000 / 801 -> Aproximadamente 10, por lo tanto, ese es el límite de mejora alcanzable.

Suponga que el tiempo de ejecución de un algoritmo secuencial es de 10000 unidades de tiempo, de las cuales 95% corresponden a código paralelizable. ¿Cuál es el límite en la mejora que puede obtenerse paralelinzado el algoritmo?

El límite de mejora se consigue utilizando 9500 procesadores los cuales ejecutan una unidad de tiempo obteniendo un tiempo paralelo de 1 unidad de tiempo + las 500 unidades de tiempo del código secuencial, siendo un total de 501 unidades de tiempo.
El speedup mide la mejora del tiempo obtenida con un algoritmo paralelo comparándola con el secuencial. Speedup = TS/TP = 10000/501 = 19,9 ~ 20. Aunque se utilicen más procesadores el mayor speedup alcanzable es este, cumpliéndose así la Ley de Amdahl que establece que para todo problema hay un límite de paralelizacion, dependiendo el mismo no de la cantidad de procesadores, sino de la cantidad de código secuencial.

Indique los posibles valores finales de x en el siguiente programa (justifique claramente su respuesta):

```
int x = 3; sem s1 = 1, s2 = 0;
   co   P(s1); x = x * x ; V(s1);                  # P1
     // P(s2); P(s1); x = x * 3; V(s1):          # P2
     // P(s1); x = x - 2; V(s2); V(s1);          # P3
   oc
```

s1 → *mutex*: puede pasar un solo proceso por vez.
s2 → *semáforo de señalización*: esperan señalización de un evento y pasa sólo uno.

P1 y P3 comienzan esperando a s1. Por ser un mutex, sólo puede continuar uno de ellos y no será interrumpido por el otro hasta liberar a s1.

1. - Si comienza P1: Asigna x=9, luego incrementa s1 permitiendo que continúe P3. P3 asigna x=7 y señala s2. Esto habilita a P2 que estaba esperando. Si P2 continúa, intentará obtener s1 con lo cual se vuelve a bloquear volviendo el control a P3. En cualquier caso, P3 libera a s1 y termina. P2 es despertado, asigna x = 21 y termina. Valor final x=21.
2. - Si comienza P3: Asigna x = 1 y señala a s2. Esto habilita a P2 que estaba esperando. Si P2 continúa, intentará obtener s1 con lo cual se vuelve a bloquear volviendo el control a P3. Cuando P3 libera a s1, P1 y P2 pueden competir por él:

1 -Si gana P1: asigna x=1, libera a s1 y termina; finaliza P2 y asigna x = 3. Valor final x=3.
2 -Si gana P2: asigna x=3, libera a s1 y termina; finaliza P1 y asigna x = 9. Valor final x=9.

P2 nunca puede comenzar la historia ya que espera un semáforo de señalización que sólo P3 señala. Cualquier historia en la que P2 esté antes de P3 es inválida.
En todas las historias los semáforos terminan con los mismos valores con los que están inicializados.

c) Dado el siguiente programa concurrente indique cuáles valores de K son posibles al finalizar, y describa una secuencia de instrucciones para obtener dicho resultado:

Process P1{ fa i=1 to K → N=N+1 af} Process P2{ fa i=1 to K → N=N+1 af }

i) 2K
ii) 2K+2
iii) K
iv) 2

i)2K -> VALOR POSIBLE (Si ejecuta todo el ForAll P1 y luego todo el ForAll P2)
ii)2K+2 -> VALOR IMPOSIBLE
iii) K -> VALOR POSIBLE (Si ejecutan al mismo tiempo P1 y P2)
iv) 2 -> VALOR POSIBLE (Si P1 carga la variable, P2 hace ForAll de k-1, luego P1 realiza el incremento de n y luego sucede lo mismo para P2)

Suponga que N procesos poseen inicialmente cada uno un valor. Se debe calcular la suma de todos los valores y al finalizar la computación todos deben conocer dicha suma.

Analice (desde el punto de vista del número de mensajes y la performance global) las soluciones posibles con memoria distribuida para arquitecturas en Estrella (centralizada), Anillo Circular, Totalmente Conectada y Árbol.

Arquitectura en estrella (centralizada)

En este tipo de arquitectura todos los procesos (workers) envían su valor local al procesador central (coordinador), este suma los N datos y reenvía la información de la suma al resto de los procesos. Por lo tanto se ejecutan 2(N-1) mensajes. Si el procesador central dispone de una primitiva broadcast se reduce a N mensajes.

En cuanto a la performance global, los mensajes al coordinador se envían casi al mismo tiempo. Estos se quedaran esperando hasta que el coordinador termine de computar la suma y envié el resultado a todos.

```
chan valor(INT),
resultados[n](INT suma);

Process P[0]{       #coordinador, v esta inicializado
    INT v;  INT sum=0;
    sum = sum+v;
    for [i=1 to n-1]{
        receive valor(v);
        sum=sum+v;
    }
    for [i=1 to n-1]
        send resultado[i](sum);
}

process P[i=1 to n-1]{  #worker, v esta inicializado
    INT v; INT sum;
    send valor(v);
    receive resultado[i](sum);
}
```

Anillo circular

Se tiene un anillo donde P[i] recibe mensajes de P[i-1] y envía mensajes a P[i+1]. P[n-1] tiene como sucesor a P[0]. El primer proceso envía su valor local ya que es lo único que conoce.

Este esquema consta de dos etapas:

1. Cada proceso recibe un valor y lo suma con su valor locar, transmitiendo la suma local a su sucesor.
2. Todos reciben la suma global.

P[0] debe ser algo diferente para poder “arrancar” el procesamiento: debe envía su valor local ya que es lo único que conoce. Se requeriran 2(n-1) mensajes.

A diferencia de la solución centralizada, esta reduce los requerimientos de memoria por proceso

```
chan valor[n](sum);
process p[0]{
    INT v; INT suma = v;
    send valor[1](suma);
    receive valor[0](suma);
    send valor[1](suma);
}

process p[i = 1 to n-1]{
    INT v; INT suma;
    receive valor[i](suma);
    suma = suma + v;
    send valor[(i+1) mod n](suma);
    receive valor[i](suma);
    if (i < n-1)
        send valor[i+1](suma);
}
```

pero tardara más en ejecutarse, por más que el número de mensajes requeridos sea el mismo. Esto se debe a que cada proceso debe esperar un valor para computar una suma parcial y luego enviársela al siguiente proceso; es decir, un proceso trabaja por vez, se pierde el paralelismo.

Totalmente conectada (simetrica)

```
chan valor[n](INT);
process p[ i=0 to n-1]{
    INT v;
    INT nuevo, suma = v;

    for [k=0 to n-1 st k <> i]
        send valor[k](v);
    for [k=0 to n-1 st k <> i]{
        receive valor[i](nuevo);
        suma = suma+nuevo;
    }
}
```

Todos los procesos ejecutan el mismo algoritmo. Existe un canal entre cada par de procesos.

Cada uno transmite su dato local v a los n-1 restantes. Luego recibe y procesa los n-1 datos que le faltan, de modo que en paralelo toda la arquitectura está calculando la suma total y tiene acceso a los n datos.

Se ejecutan n(n-1) mensajes. Si se dispone de una primitiva de broadcast, serán n mensajes. Es la solución más corta y sencilla de programar, pero utiliza el mayor número de mensajes si no hay broadcast.

Arbol

Se tiene una red de procesadores (nodos) conectados por canales de comunicación bidireccionales. Cada nodo se comunica directamente con sus vecinos. Si un nodo quiere enviar un mensaje a toda la red, debería construir un árbol de expansión de la misma, poniéndose a el mismo como raíz.

El nodo raíz envía un mensaje por broadcast a todos los hijos, junto con el árbol construido. Cada nodo examina el árbol recibido para determinar los hijos a los cuales deben reenviar el mensaje, y así sucesivamente.

Se envían n-1 mensajes, uno por cada padre/hijo del árbol.

Implemente una solución al problema de exclusión mutua distribuida entre N procesos utilizado un algoritmo de tipo token passing con mensajes asincrónicos.

El algoritmo de token passing, se basa en un tipo especial de mensaje o “token” que puede utilizarse para otorgar un permiso (control) o para recoger información global de la arquitectura distribuida.

Si User[1:n] son un conjunto de procesos de aplicación que contienen secciones críticas y no críticas. Hay que desarrollar los protocolos de interacción (E/S a las secciones críticas), asegurando exclusión mútua, no deadlock, evitar demoras innecesarias y eventualmente fairness.

Para no ocupar los procesos User en el manejo de los tokens, ideamos un proceso auxiliar (helper) por cada User, de modo de manejar la circulación de los tokens. Cuando helper[i] tiene el token adecuado, significa que User[i] tendrá prioridad para acceder a la sección crítica.

```
chan token[n]();                        # para envio de tokens
chan enter[n](), go[n](), exit[n]();    # para comunicación proceso-helper


process helper[i = 1..N] {
    while(true){
        receive token[i]();             # recibe el token
        if(!(empty(enter[i]))){         # si su proceso quiere usar la SC
            receive enter[i]();
            send go[i]();               # le da permiso y lo espera a que termine
            receive exit[i]();
        }
        send token[i MOD N +1]();       # y lo envía al siguiente cíclicamente
    }
}

process user[i = 1..N] {
    while(true){
        send enter[i]();
        receive go[i]();
        ... sección crítica ...
        send exit[i]();
        ... sección no crítica ...
    }
}
```

(Broadcast atómico). Suponga que un proceso productor y n procesos consumidores comparten un buffer unitario. El productor deposita mensajes en el buffer y los consumidores los retiran. Cada mensaje depositado por el productor tiene que ser retirado por los n consumidores antes de que el productor pueda depositar otro mensaje en el buffer.

a) Desarrolle una solución utilizando semáforos.

```
sem depositar:=1;
sem retirar:= 1;
sem consumir[n]:=([n] 0);
int cant_consumido:= ([n] 0);
T buffer;

process productor{
    while(true){
        P(depositar);
            buffer:= generarDato();  //Devuelve un entero para el buffer
            cant_consumido:= 0;
            for i to n do
                V(consumi[i])
    }
}

process consumidor[i: 1..n]{
    T dato;
    while(true){
        P(consumi[i])            //Espero que el dato este en el buffer
            P(retirar)               //Espero para tener acceso al buffer
                dato:=buffer;
                cant_consumido ++;
                if (cant_consumido == n)
                    V(depositar);
            V(retirar);
    }
}
```

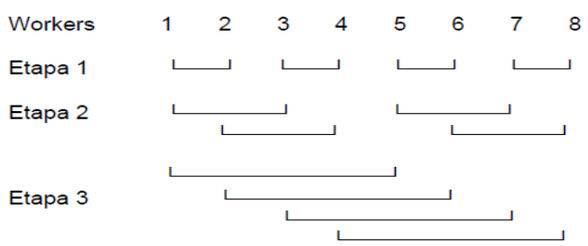
b) Suponga que el buffer tiene b slots. El productor puede depositar mensaje sólo en slots vacíos y cada mensaje tiene que ser recibido por los n consumidores antes de que el slot pueda ser reusado. Además, cada consumidor debe recibir los mensajes en el orden en que fueron depositados (note que los distintos consumidores pueden recibir los mensajes en distintos momentos siempre que los reciban en orden). Extienda la respuesta dada en a) para resolver este problema

más general.

```
sem retirar[tamBuffer]:= 1;                //semaforo para cada slot del buffer
sem consumir[n]:=([n] 0);
int cant_consumido[tamBuffer]:=([n] 0);
T   buffer[tamBuffer];

process productor{
    int posLibre:= 0;              //Siguiente posiscion libre del buffer (productor)
    while(true){
        P(depositar);
            buffer[posLibre]:=generarDato();   //Devuelve dato tipo T para el buffer
            cant_consumido[posLibre]:= 0;
            for i to n
                V(consumir[i]);
            posLibre:= (posLibre + 1) mod n;
    }
}

process Consumidor[i: 1..n]{
    int post_actual:=0;                     //Slot del que debe consumir
    T dato;
    while(true){
        P(consumir[i]);
            P(retirar[post_actual])         //Espera por un slot en particular
                dato:=buffer;
                cant_consumido[post_actual] ++;
                if (cant_consumido[post_actual] == n)
                    V(depositar);
            V(retirar[post_actual]);
            post_actual[i]= (post_actual[i] + 1) mod n;
    }
}
```

Implemente una butterfly barrier para 8 procesos usando variables compartidas.

Una butterfly barrier tiene log2n etapas. Cada Worker sincroniza con un Worker distinto en cada etapa. En particular, en la etapa s un Worker sincroniza con un Worker a distancia 2^(s-1). Se usan distintas variables flag para cada barrera de dos procesos. Cuando cada Worker pasó a través de log2n etapas, todos los Workers deben haber arribado a la barrera y por lo tanto todos pueden seguir. Esto es porque cada Worker ha sincronizado directa o indirectamente con cada uno de los otros.



```
int N =8;
int E=log(N);
int arribo[1:N] = ([n] 0);

process Worker[i = 1 to n] {
    int j;
    int resto;
    int distancia;

    while (true) {
        //Sección de código anterior a la barrera.
        //Inicio de la barrera

        for (etapa = 1; etapa<=E; etapa++] {
            distancia = 2 ^(s-1);
            resto = (i mod 2^s);
            if (resto=0)or(resto>distancia) //Ejemplo, 7 con 8 ronda 1, 1 con 5 ronda 3
                distancia = -distancia;
            j = i + distancia;
            while (arribo[i] == 1) skip;
              arribo[i] = 1;
            while (arribo[j] == 0) skip;
              arribo[j] = 0;
        }
        //Fin de la barrera
        //Sección de código posterior a la barrera.
    }
}
```

Suponga n^2 procesos organizados en forma de grilla cuadrada. Cada proceso puede comunicarse solo con los vecinos izquierdo, derecho, de arriba y de abajo (los procesos de las esquinas tienen solo 2 vecinos, y los otros en los bordes de la grilla tienen 3 vecinos). Cada proceso tiene inicialmente un valor local v.

a) **Escriba un algoritmo heartbeat que calcule el máximo y el mínimo de los n2 valores. Al terminar el programa, cada proceso debe conocer ambos valores. (Nota: no es necesario que el algoritmo esté optimizado).**

```
Chan valores[1:n;1:n](int);

Process P[i = 1 to n, j = 1 to n]{
    Int v;
    Int Nuevo, minimo=v, maximo=v;
    Int cantVecinos;
    Vecinos[1..cantVecinos]

    For [k = 1 to cantGeneraciones]{
        For [p = 1 to cantVecinos]{
            Send valores[ vecimos[p].fila,vecimos[p].columna](v)
        }
        For [p = 1 to cantVecinos]{
            Receive valores[i,j](nuevo)
            if nuevo < minimo
                minimo= nuevo
            if nuevo> máximo
               Máximo = nuevo

        }

    }
}
```

b) **Analice la solución desde el punto de vista del número de mensajes.**

4 procesos (las esquinas) envían solo 2 mensajes por ronda. 4*2*(n-1)*2 = (n-1)*16 mensajes

(n-2) * 4 procesos (los bordes) envían 3 mensajes por ronda. (n-2)*2*3*(n-1)*2 = (n-1)(n-2)*12 mensajes.

(n-2)*(n-2) procesos envían 4 mensajes por ronda. $(n-2)^2*4*(n-1)*2 = (n-2)^2*(n-1)*8$.

c) **¿Puede realizar alguna mejora para reducir el número de mensajes?**
No, no existe una manera de determinar cuándo un proceso obtuvo el mínimo o el máximo por lo que es necesario para asegurarse que todos los procesos tienen los valores globales de mínimo y máximo que se ejecuten las (n-1)*2.

Suponga que una imagen se encuentra representada por una matriz a (nxn), y que el valor de cada pixel es un número entero que es mantenido por un proceso distinto(es decir, el valor del pixel I,J, esta en el proceso P(I,J). Cada proceso puede comunicarse solo con su vecinos izquierdo, derecho, arriba y abajo.(los procesos de los esquinas tienen solo 2 vecinos, y los otros bordes de la grilla tienen 3 vecinos) .

a) **Escriba un algoritmo Herbeat que calcule el maximo y el minimo valor de los pixels de la imagen. Al terminar el programa, cada proceso debe conocer ambos valores.**

```
chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool, max : int, min : int);

process nodo[p = 1..n] {
    bool vecinos[1:n];                  # inicialmente vecinos[q] true si q es vecino de p
    bool activo[1:n] = vecinos;         # vecinos aún activos
    bool top[1:n,1:n] = ([n*n]false);   # vecinos conocidos (matriz de adyacencia)
    bool nuevatop[1:n,1:n];
    int r = 0;
    bool listo = false;
    int emisor;
    bool qlisto;
    int miValor, max, min;          # miValor inicializado con el valor del pizel
    top[p,1..n] = vecinos;          # llena la fila para los vecinos
    max := miValor; min:= miValor;

    while(not listo) {                      # envía conocimiento local de la topología a
        for[q = 1 to n st activo[q] ]       # sus vecinos
            send topologia[q](p, false, top, max, min);
        for [q = 1 to n st activo[q] ] {
            receive topologia[p](emisor,qlisto,nuevatop, nuevoMax, nuevoMin);
            top = top or nuevatop;                  # recibe las topologías y hace OR con su top juntando la informacion
            if(nuevoMax>max) nuevoMax := max;     # actualiza los maximos y minimos
            if(nuevoMin<min) nuevoMin := min;
            if(qlisto) activo[emisor] = false;
        }
        if(todas las filas de top tiene 1 entry true) listo = true;
        r := r + 1;
    }
    # envía topología completa a todos sus vecinos aún activos
    for[q = 1 to n st activo[q] ]
        send topologia[q](p,listo,top, max, min);
    # recibe un mensaje de cada uno para limpiar el canal
    for [q=1 to n st activo[q]]
        receive topologia[p](emisor,d,nuevatop, nuevoMax, nuevoMin);
}
```

j

b) **Analice la solución de desde el punto de vista del número de mensajes.**

Si M es el numero maximo de vecinos que puede tener un nodo, y D es el diametro de la red, el número de mensajes maximo que pueden intercambiar es de 2n * m * (D+1). Esto es porque cada nodo ejecuta a lo sumo D-1 rondas, y en cada una de ellas manda 2 mensajes a sus m vecinos.

c) **Puede realizar alguna mejora para reducir el número de mensajes.**

El algoritmo centralizado requiere el intercambio de 2n mensajes, uno desde cada nodo al server central y uno de respuesta. El algoritmo descentralizado requiere el intercambio de más mensajes. Si m y D son relativamente chicos comparados con n, entonces el número de mensajes no es mucho mayor que para el algoritmo centralizado. Además, estos pueden ser intercambiados en paralelo en muchos casos, mientras que un server centralizado espera secuencialmente recibir un mensaje de cada nodo antes de enviar cualquier respuesta.

Dado el siguiente programa concurrente, indique cuál es la respuesta correcta (justifique claramente):

```
int a = 1, b = 0;
co ⟨await (b = 1) a = 0 ⟩
 // while (a = 1) { b = 1; b = 0; }
oc
```

a) Siempre termina
b) Nunca termina
c) Puede terminar o no

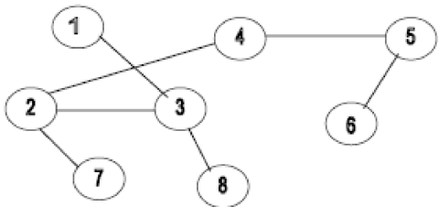
Desde el punto de vista conceptual con una política fuertemente fair, el programa eventualmente termina, porque b se vuelve infinitamente 1. Sin embargo con una política débilmente fair, el programa puede no terminar, porque b es también infinitamente 0. Desafortunadamente, es imposible idear un procesador con una política de scheduling que sea práctica y fuertemente fair. Por ejemplo, RR y el time slicing son prácticos pero no fuertemente fair, porque, en general, los procesos ejecutan en orden impredecible.
Un scheduler multiprocesador que ejecute los procesos en paralelo también es práctico, pero no es fuertemente fair. Esto se debe a que el segundo proceso siempre puede examinar b cuando es 0.

Resuelva el problema de encontrar la topología de una red utilizando mensajes asincrónicos. Muestre con un ejemplo la evolución de la matriz de adyacencia para una red con al menos 7 nodos y de diámetro al menos 4. Compare conceptualmente con una solución utilizando PMS.

```
chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool)

Process Nodo[p:1..n] {
    bool vecinos[1:n];# inicialmente vecinos[q] true si q es vecino de Nodo[p]
    bool activo[1:n] = vecinos # vecinos aún activos
    bool top[1:n,1:n] = ([n*n]false) # vecinos conocidos
    int r = 0; bool listo = false;
    int emisor; bool qlisto; bool nuevatop[1:n,1:n];
    top[p,1..n] = vecinos; # llena la fila para los vecinos

    while (not listo) {
        # envía conocimiento local de la topología a sus vecinos
        for [q = 1 to n st activo[q] ] send topologia[q](p,false,top);
        # recibe las topologías y hace or con su top
        for [q = 1 to n st activo[q] ] {
        receive topologia[p](emisor,qlisto,nuevatop);
        top = top or nuevatop;
        if (qlisto) activo[emisor] = false;
        }
        if (todas las filas de top tiene 1 entry true) listo=true;
        r := r + 1
    }

    # envía topología a todos sus vecinos aún activos
    for [q = 1 to n st activo[q] ] send topologia[q](p,listo,top);
    # recibe un mensaje de cada uno para limpiar el canal
    for [q=1 to n st activo[q]] receive topologia[p](emisor,d,nuevatop)
}
```

Representamos cada nodo por un proceso Nodo[p:1..n].Dentro de cada proceso, podemos representar los vecinos de un nodo por un vector booleano vecinos[1:n]con el elemento vecinos[q] true en Nodo[p]si q es vecino de p. Estos vectores se asumen inicializados con los valores apropiados. La topología final top puede ser representada por una matriz de adyacencia, con top:[1:n,1:n]true si p y q son nodos vecinos.

Después de r rondas, el nodo p conocerá la topología a distancia r de él. En particular, para cada nodo q dentro de la distancia r de p, los vecinos de q estarán almacenados en la fila q de top. Dado que la red es conectada, cada nodo tiene al menos un vecino. Así, el nodo p ejecutó las suficientes rondas para conocer la topología tan pronto como cada fila de top tiene algún valor true. En ese punto, p necesita ejecutar una última ronda en la cual intercambia la topología con sus vecinos; luego p puede terminar. Esta última ronda es necesaria pues p habrá recibido nueva información en la ronda previa. También evita dejar mensajes no procesados en los canales. Dado que un nodo podría terminar una ronda antes (o después) que un vecino, cada nodo también necesita decirle a sus vecinos cuando termina. Para evitar deadlock, en la última ronda un nodo debería intercambiar mensajes solo con sus vecinos que no terminaron en la ronda previa.



Inicialmente en el nodo 3:

	1	2	3	4	5	6	7	8
1								
2								
3	T	T	T					T
4								
5								
6								
7								
8								

Después de una ronda, en nodo 3:

	1	2	3	4	5	6	7	8
1	T		T					
2		T	T	T			T	
3	T	T	T					T
4								
5								
6								
7								
8			T					T

Después de dos rondas, en nodo 3:

	1	2	3	4	5	6	7	8
1	T		T					
2		T	T	T			T	
3	T	T	T					T
4		T		T	T			
5								
6								
7		T					T	
8			T					T

PARA MI ACA DEBERIA SEGUIR LA RONDA HASTA LLEGAR HASTA 4 QUE ES EL DIAMETRO.

El proceso del broadcast es un proceso asincrónico, resolverlo con PMS causa que se reduzca la eficiencia.

a. Que mecanismo de pasaje de mensaje es más adecuado para la resolución? Justifique claramente.

El mecanismo de pasaje de mensajes asincrónico es el más adecuado para la resolución junto con algoritmos heartbeat ya que nos permite una interacción entre procesos de manera que cada procesador es modelizado por un proceso y los links de comunicación con canales compartidos. En particular, cada proceso ejecuta una secuencia de iteraciones. En cada iteración, un proceso envía su conocimiento local de la topología a todos sus vecinos, luego recibe la información de ellos y la combina con la suya. La computación termina cuando todos lo procesos aprendieron la topología de la red entera. El criterio de terminación no siempre puede ser decidida localmente.